SSC0300 - Linguagens de Programação e Aplicações

Aula02: Comandos Básicos do C

Professor: Eduardo do Valle Simões
Email: simoes@icmc.usp.br

Objetivo: Revisar os conceitos básicos de programação em Linguagem C. Serão revistos:

a. Input and Output (I/O):stdio.h
b. Variables and Constants
c. Conditionals
d. Looping and Iteration
e. Arrays
f. Strings <string.h>
g. Structures
h. Pointers

1) Input and Output (I/O):stdio.h

Output: printf

%c -- characters
%d -- integers
%f -- floats

ex.:
```
printf("Um Caractere%c, Um Inteiro%d, e Um Ponto Flutuante%f", ch, i, x);
```

Input: scanf

ex.:
```
int a;
long int b;
unsigned int c;
float d;
double e;
long double f;
char s[100];

scanf("%d", &a);  // store an int
scanf(" %d", &a); // eat any whitespace, then store an int
scanf("%s", s); // store a string
scanf("%Lf", &f); // store a long double

// store an unsigned, read all whitespace, then store a long int:
scanf("%u %ld", &c, &b);

// store an int, read whitespace, read "blendo", read whitespace,
// and store a float:
```

```
scanf("%d blendo %f", &a, &d);

// read all whitespace, then store all characters up to a newline
scanf(" %[^\n]", s);

// store a float, read (and ignore) an int, then store a double:
scanf("%f %*d %lf", &d, &e);

// store 10 characters:
scanf("%10c", s);
```

Formatação:

`%[flags][width][.precision][length]specifier`

specifier define o tipo e a interpretação recebida pelo argumento:

specifier	**Output**	**Example**
`C`	Character	`a`
`d` or `i`	Signed decimal integer	`392`
`E`	Scientific notation (mantise/exponent) using `e` character	`3.9265e+2`
`E`	Scientific notation (mantise/exponent) using `E` character	`3.9265E+2`
`F`	Decimal floating point	`392.65`
`G`	Use the shorter of `%e` or `%f`	`392.65`
`G`	Use the shorter of `%E` or `%f`	`392.65`
`O`	Signed octal	`610`
`S`	String of characters	`sample`
`U`	Unsigned decimal integer	`7235`
`X`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (capital letters)	`7FA`
`p`	Pointer address	`B800:0000`
`n`	Nothing printed. The argument must be a pointer to a signed `int`, where the number of characters written so far is stored.	
`%`	A `%` followed by another `%` character will write `%` to the stream.	

Os outros campos são opcionais:

Flags	**Description**
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.

#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see *width* sub-specifier).

Width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**Description**
.number	For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For e, E and f specifiers: this is the number of digits to be printed **after** de decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type: it has no effect. When no *precision* is specified, the default is 1. If the period is specified without an explicit value for *precision*, 0 is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

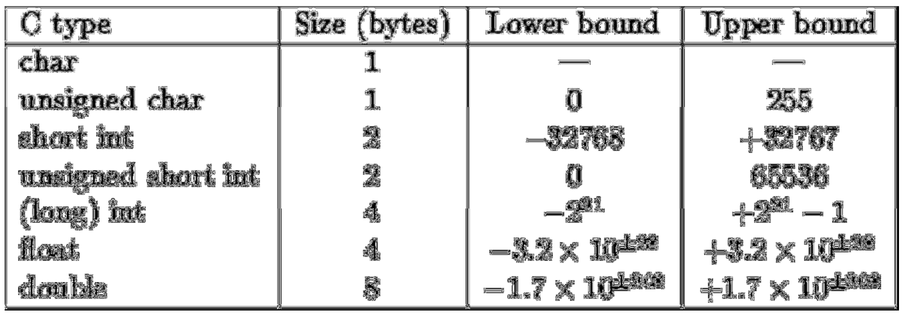
Length	**Description**
H	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
L	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

Escape Sequence: Incluídas dentro do campo de texto. Ex.: printf("\n\n Alerta!! \n\n");

Escape Sequence	Name	Meaning
\a	Alert	Produces an audible or visible alert.
\b	Backspace	Moves the cursor back one position (non-destructive).
\f	Form Feed	Moves the cursor to the first position of the next page.
\n	New Line	Moves the cursor to the first position of the next line.
\r	Carriage Return	Moves the cursor to the first position of the current line.
\t	Horizontal Tab	Moves the cursor to the next horizontal tabular position.
\v	Vertical Tab	Moves the cursor to the next vertical tabular position.
\'		Produces a single quote.
\"		Produces a double quote.
\?		Produces a question mark.
\\		Produces a single backslash.
\0		Produces a null character.
\ddd		Defines one character by the octal digits (base-8 number). Multiple characters may be defined in the same escape sequence, but the value is implementation-specific (see examples).
\xdd		Defines one character by the hexadecimal digit (base-16 number).

2) Variables

Tipos de Dados em C:



C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	-32768	$+32767$
unsigned short int	2	0	65536
(long) int	4	-2^{31}	$+2^{31}-1$
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

Declarando Variáveis Globais:

```
unsigned char number, sum1=sum2=5;

int bignumber=5, bigsum;

float x = 3.0;

char text1, letter = 'A';

main()

 {

 }
```

3) Constants

```
#define inicio 50

int const a = 1;
const int a = 2;
```

OBS.: Aritmética Simples:

a) x=((++z)-(w--)) / 100; z++; x=(z-w) / 100; w--;
b) x = 3 / 2 is 1 even if x is declared a float!!
c) RULE: If both arguments of / are integer then do integer division
d) Forma correta de dividir floats: x = 3.0 / 2 or x= 3 / 2.0 or (better) x = 3.0 / 2.0
e) Shorthands: i = i + 3 → i += 3 x = x*(y + 2) → x *= y + 2

f) Prioridades:
```
( )  [  ]  -> .
!  ~  - * & sizeof cast ++ -
      (these are right->left)
* / %
+ -
< <= >= >
== !=
&
^               |
&&
||
?:        (right->left)
= += -= (right->left)
,    (comma)
```

Exemplo:

```
 a < 10 && 2 * b < c
```

é interpretado como: `( a < 10 ) && ( ( 2 * b ) < c )`

4) Conditionals

Cuidado: if (i = j) faz i = j e depois retorna o valor de j e resulta em 1 (true) se j != 0

O correto é: if (i == j)

Not equal is: !=

Logical Operators: && for logical AND, || for logical OR.
Cuidado: & and | have a different meaning for bitwise AND and OR

Other operators < (less than) , > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

Sintaxe do IF:
```
        if  (expression)
                statement;
...ou:
        if  (expression)
                statement1;
         else
                statement2;
...ou:
        if  (expression)
                statement1;
        else if (expression)
                statement2;
        else
                statement3;
```

Obs.: se usar mais de um statement, colocar entre { }

Ex.:
```
if  (expression)
        {
        statement1;
        statement2;
        statement3;
        }
```

The ? operator

expression1 ? expression2: expression3

significa: if expression1 then expression2 else expression3

Ex.:
```
z = (a>b) ? a : b;
```

significa:
```
if (a>b)
        z = a;
else
        z=b;
```

The switch statement

```
switch (expression) {
                case item1:
                        statement1;
                        break;
                case item2:
                        statement2;
                        break;
                        . . .
                case itemN:
                        statementn;
                        break;
                default:
                        statement;
                        break;
                }
```

Obs.: Se não usar **break**, todos os outros casos serão avaliados!!!
The **default** case is optional and catches any other cases.

Ex.:

```
switch (letter)
              {
               case ‘A’:      numberofvowels++;

               case ‘E’:      numberofvowels++;

               case ‘I’:      numberofvowels++;

               case ‘O’:      numberofvowels++;

               case ‘U’:      numberofvowels++;


               case ‘ ’:       numberofspaces++;

               default:        numberofconstants++;
              }
```

5) Looping and Iteration

The for statement

```
for  (expression1; expression2; expression3)
                          statement;
                          or {block of statements}
```

expression1 initialises; *expression2* is the terminate test; *expression3* is the modifier

Ex.:

```
for (i=0;i<=10;i++)

for (x=3;x>0;x--)

for (x=4;((x>3) && (x<9)); x++)

 for (x=4,y=0;((x>3) && (y<9)); x++,y+=2)

 for (x=0,y=4,z=4000;z; z/=10)
```

The while statement

```
while (expression)
            statement;
            or {block of statements}
```

Ex.:

```
while (x>0)
        { printf("x=%d", x);
           x--;   }
while (i++ < 10);
while ( (ch = getchar()) != `q')
        putchar(ch);
```

The do-while statement

```
do
   statement;
while (expression);
```

Ex.:

```
do {
        printf("x = %d",x--);
    }
while (x>0);
```

Loop Control

C provides two commands to control how we loop:

> **break** -- exit form loop or switch.
> **continue** -- skip 1 iteration of loop.

Ex.:

```
while (scanf( ``%d", &value ) == 1 && value != 0) {

                  if (value < 0) {
                          printf(``Illegal value \n");
                          break;
                          /* Abandon the loop */
                  }

                  if (value > 100) {
                          printf(``Invalid value\n");
                          continue;
                          /* Skip to start loop again */
                  }

                  /* Process the value read */
                    /* guaranteed between 1 and 100 */
                      ....;

                  ....;
        } /* end while value = 0 */
```

6) Arrays

Exemplo de array em C:

```
int numbers[50];
```

Cuidado: Em C, isso significa numbers[0] até numbers[49] !!!!!

Como acessar os elementos:

```
Thirdnumber = numbers[2];
numbers[5]=100;
```

Multi-dimensional arrays:

```
int tableofnumbers[50][50];

int bigD[50][50][40][30]......[50];

        anumber = tableofnumbers[2][3];
        tableofnumbers[25][16] = 100;
```

Obs.: char Aname[10][20];

* access elements via **20*row + col + base_address** in memory.

Programa para preencher e imprimir uma Matriz:

```
unsigned char V[10];
int i;

for (i=0;i<10;i++)
        {
        V[i] = i;
}

for (i=0;i<10;i++)
        {
        printf("V[%d] = %d\n", i, V[i]);
}
```

7) Strings

Em C, strings são definidas como arrays de caracteres. Ex.:

```
char name[50];
```

Cuidado: C não tem suporte a manipulação de strings !!! Portando, os comandos a seguir são ilegais:

```
char firstname[50],lastname[50],fullname[100];

        firstname= "Arnold"; /* Illegal */
        lastname= "Schwarznegger"; /* Illegal */
        fullname= "Mr"+firstname+lastname; /* Illegal */
```

Para declarer uma string, temos que trata-la como Array !!

```
char name[]="simoes";

printf("%s", name);
```

Para manipular strings, deve-se usar a library <string.h>.

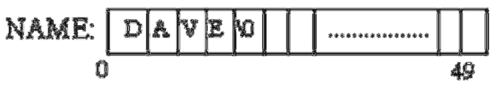
Para imprimir uma string, usa-se printf com um caracter de controle especial: **%s**. Ex.:

```
printf("%s", name);
```

Obs.: Só é necessário dar o nome da string

Para se trabalhar com strings de tamanho variado, se usa o caracter **\0** para indicar o final da string.

Então, se tivermos uma string: char NAME[50]; e guardarmos "DAVE" nela, o resultado final sera:



a) Manipulação de Strings

#include <string.h>

Descrição das funções mais comuns:

int strcmp(const char *string1,const char *string2) - Compare string1 and string2 to determine alphabetic order.

int strncmp(const char *string1, char *string2, size_t n) -- Compare first n characters of two strings.

int strcasecmp(const char *s1, const char *s2) -- case insensitive version of strcmp().
int strncasecmp(const char *s1, const char *s2, int n) -- case insensitive version of strncmp().

char *stpcpy (const char *dest,const char *src) -- Copy one string into another.
char *strcpy(const char *string1,const char *string2) -- Copy string2 to stringl.
Char *strncpy(const char *string1,const char *string2, size_t n) -- Copy first n characters of string2 to stringl .

int strlen(const char *string) -- Determine the length of a string.

char *strncat(const char *string1, char *string2, size_t n) -- Append n characters from string2 to stringl.

Examplo:

```
char *str1 = "HELLO";
char *str2;
int length;

length = strlen("HELLO"); /* length = 5 */

strcpy(str2,str1); /* str2 = str1 */
```

b) Comparação

strcmp() compara lexicamente duas strings e retorna:

Less than zero -- if string1 is lexically less than string2
Zero -- if string1 and string2 are lexically equal
Greater than zero -- if string1 is lexically greater than string2

c) Cópias de pedaços:

strncat(), strncmp,() and strncpy() copiam somente o número de caracteres especificado:

```
char *str1 = "HELLO";
char *str2;
int length = 2;


strcpy(str2,str1, length); /* str2 = "HE" */
```

Cuidado : str2 is **NOT NULL TERMINATED!!**

d) Busca em Strings

char *strchr(const char *string, int c) -- Find first occurrence of character c in string.
char *strrchr(const char *string, int c) -- Find last occurrence of character c in string.
char *strstr(const char *s1, const char *s2) -- locates the first occurrence of the string s2 in string s1.
char *strpbrk(const char *s1, const char *s2) -- returns a pointer to the first occurrence in string s1 of any character from string s2, or a null pointer if no character from s2 exists in s1
size_t strspn(const char *s1, const char *s2) -- returns the number of characters at the begining of s1 that match s2.
size_t strcspn(const char *s1, const char *s2) -- returns the number of characters at the begining of s1 that do not match s2.
char *strtok(char *s1, const char *s2) -- break the string pointed to by s1 into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by s2.
char *strtok_r(char *s1, const char *s2, char **lasts) -- has the same functionality as strtok() except that a pointer to a string placeholder lasts must be supplied by the caller.

Exemplo1:

```
char *str1 = "Hello";
char *ans;

ans = strchr(str1,'l');          // ans points to the location str1 + 2

ans = strpbrk(str1,'aeiou');     // ans points to the location str1 + 1 (the first e).

ans = strstr(str1,'lo');         // ans = str + 3.
```

e) Testando os Caracteres

```
#include <ctype.h>               // contém funções para converter e testar caracteres
```

int isalnum(int c) -- True if c is alphanumeric.
int isalpha(int c) -- True if c is a letter.
int isascii(int c) -- True if c is ASCII .
int iscntrl(int c) -- True if c is a control character.
int isdigit(int c) -- True if c is a decimal digit
int isgraph(int c) -- True if c is a graphical character.
int islower(int c) -- True if c is a lowercase letter
int isprint(int c) -- True if c is a printable character

int ispunct (int c) -- True if c is a punctuation character.
int isspace(int c) -- True if c is a space character.
int isupper(int c) -- True if c is an uppercase letter.
int isxdigit(int c) -- True if c is a hexadecimal digit

Character Conversion:

toascii(int c) -- Convert c to ASCII .
tolower(int c) -- Convert c to lowercase.
toupper(int c) -- Convert c to uppercase.

8) Structures

```
struct gun      {       char name[50];
                        int magazinesize;
                        float calibre;
                };

struct gun armas1;

struct gun armas2={"Uzi",30,7};
```

Ou:

```
struct gun              {       char name[50];          // a tag gun é opcional !!
                                int magazinesize;
                                float calibre;
                        } arma;
```

Para acessar um membro:

```
  arma.magazinesize=100;

int a;

a= arma.magazinesize;

arma.calibre = a*5.5;

printf("%d", arma.magazinesize);
```

a) Definindo novos tipos de dados

```
typedef struct gun      {       char name[50];                // a tag gun é opcional !!
                                int magazinesize;
                                float calibre;
                        } tipogun;

tipogun arma={"Uzi",30,7};
```

b) Arrays de Structures

```
tipogun armas[1000];
```

Para acessar: armas[50].calibre=100;

9) Pointers

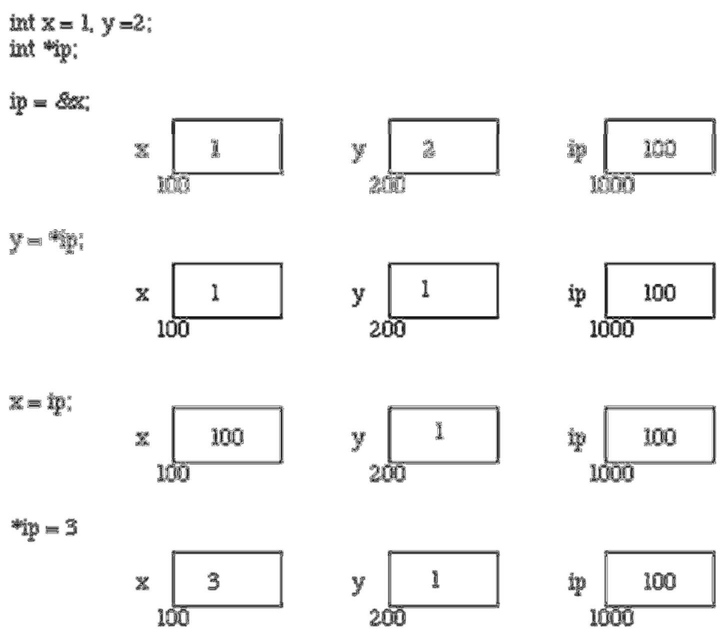
Um ponteiro é uma variável . . .
    . . . Que contém o endereço de memória de outra variável !!

➔ operator **&** gives the "address of a variable".

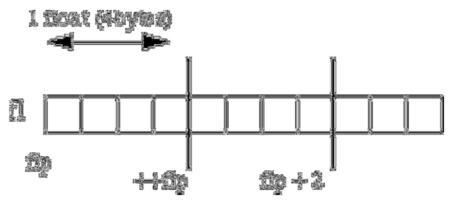
➔ operator * gives the "contents of an object pointed to by a pointer".

É preciso associar um ponteiro a um tipo: ex.: int *p

```
int x = 1, y = 2;
int *ip;

ip = &x;

y = *ip;

x = ip;

*ip = 3;
```



Importante: Quando um ponteiro é declarado, ele não aponta para nada! Temos que apontá-lo para alguma variável antes de usá-lo.



a) Ponteiros e Funções

Quando C passa argumentos para uma função, ele passa “por valor”.

Quando queremos alterar o valor de variáveis locais em uma função, devemos fazer isso explicitamente através de ponteiros.

Ex.:
swap(x, y)   WON'T WORK.

A solução está em ponteiros: Passamos o endereço das variáveis que queremos alterar...

swap(&x, &y)

```
void swap(int *px, int *py)   // Resgatamos esses endereços com ponteiros
        {
        int temp;
        temp = *px;                    // *px=contenteudo de a
        *px = *py;                     // *py=contenteudo de b
        *py = temp;
        }
```

Podemos retornar ponteiros de funções.

```
typedef struct {float x,y,z;} COORD;

main()
        {  COORD p1, *coord_fn();          // declare coord_fn to return ptr of COORD
type
        p1 = *coord_fn(...);               // assign contents of address returned
        }

COORD *coord_fn(...)
        {  COORD p2;
         p2 = ....;                        // assign structure values
         return &p2;                       // return address of p2
         }
```

b) Ponteiros e Arrays

Podemos usar ponteiros para apontar para arrays!

```
int a[10], x;
int *pa;
pa = &a[0];          // pa pointer to address of a[0]
x = *pa;             // x = contents of pa (a[0] in this case)

pa + i   ==   a[i]
```

Há várias formas de lidar com ponteiros para arrays:

```
Mais comum:          pa = &a[0];
                     pa = a;
                     a[i]   ==   *(a + i).
                     &a[i]   ==   a + i.
```

Obs.:   i) Um ponteiro é uma variáel: pa = a and pa++
        ii) Um array NÃO É UMA VARIÁVEL: a = pa and a++ ARE ILLEGAL

Importante: Quando um array é passado para uma função, o que é realmente passado é o endereço de seu primeiro elemento na memória!!

strlen(s)   ==   strlen(&s[0])

Por isso que a função é declarada:

```
int strlen(char s[]);
```

Uma declaração equivalente é:

```
int strlen(char *s);                // char s[]   ==   char *s.
```

strlen() é uma função da standard library que retorna o comprimento de uma string:

```
  int strlen(char *s)
                { char *p = s;
                while (*p != '\0');     // End of file   ==   '\0'
                        p++;
                return p - s;           // Endereço de p  -  endereço de s
                }
```

strcpy () é uma função da standard library que copia uma string para outra:

```
  void strcpy(char *s, char *t)
                {  while ( (*s++ = *t++) != '\0');}

                ou

                {  while (*t != '\0')
                        {
                                *s = *t;
                                s++;
                                t++;
                        };
                 *s = *t;
                }
```

c) Ponteiros e Structures

Podemos usar ponteiros para structures:

```
struct COORD {float x,y,z;} pt;

struct COORD *pt_ptr;

pt_ptr = &pt;                   // assigns pointer to pt
```

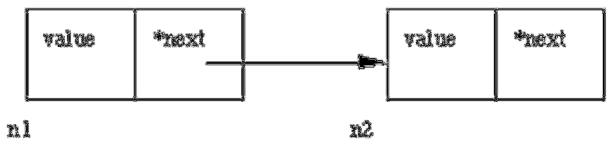
Agora, o operador -> pode ser usado para acessar um membro da structure:

```
  pt_ptr->x = 1.0;

  pt_ptr->y = pt_ptr->y - 3.0;
```

Ex.: Lista Encadeada:

```
  typedef struct {  int value; ELEMENT *next;  } ELEMENT;

ELEMENT n1, n2;
```

```
n1.next = &n2;
```


value
*next
value
*next
n1
n2